1 Introduction

The development of distributed applications is a difficult task. Although today eased by middleware there is still a need for additional tools. They should help the developer to specify the distribution structure of an application, to prepare the system for distribution and finally to insert the actual code interacting with the middleware. The first two activities preferably operate on the architecture of a software system. The architecture is naturally represented as a graph.

We regard the preparation of a system for distribution as part of an (continuous) development process, thus we support forward and reengineering as well (see also [Rad97]). Transformations of an application system can be represented as graph rewriting steps.

In section 2 we motivate the task of developing a distributed system by studying a small example program. The methodology for the distribution process and a tool we developed will be outlined in section 3. Section 4 summarizes related work.

2 Motivation

Today, communication in distributed systems is achieved by middleware products like CORBA [OMG95] or Microsoft's DCOM (COM+/DNA). These ease the development process of distributed systems considerably. The middleware abstracts from differences in operating systems and hardware (e.g. the different byte order between processors).

A system using CORBA invokes a method on a remote object via a generated stub in the local address space. This leads to a certain degree of transparency between local and remote method invocations. However, there are still semantic differences between local and remote method invocation: (1) Services might be unavailable or erroneous due to network failures, leading to new kinds of exceptions that have to be handled. (2) Remote objects can not be created using the language's new primitive. (3) For sake of efficiency, care must be taken of the handling of parameters. Should we make deep copies or pass references? This non-exhaustive list of items shows that even when using a sophisticated middleware like CORBA, a lot of questions have to be answered.

After first defining some basic notions, we will explain the problems of creating a distributed system by means of a simple example.

2.1 Basic Notions

The term architecture is used as a synonym for a description of the static system structure. In the Unified Modelling Language (UML) [Rat97] the architecture is captured by class diagrams. They show classes and interfaces of a system and the relationships between them.

A subsystem is a means of hierarchically grouping related classes and interfaces. Subsystems also encapsulate their members by restricting access to classes and interfaces with a public/private visibility attribute. Unlike in UML, we currently do not support specialization on the level of subsystems; apart from this aspect the subsystems are a subset of UMLs...
package concept.

A **component** is an *executable* part of an application. It is self-contained with respect to static references (otherwise we would not be able to link it), but has to resolve services *provided* by other components at runtime. A component consists of all participating classes and interfaces probably nested in subsystems. We require that exactly one programming language is used within the component\(^1\). The component is executed in its own (operating system) process. A class is contained in exactly one subsystem, but it can be associated to more than one component.

The definitions above are given in plain English and their meaning might be subject to different interpretations. In section 3 we specify these terms in a meta model by means of a graph schema.

### 2.2 An Example

The purpose of the following small example is to motivate the desire of a tool supporting the restructuring and distribution. Let us consider the architecture of an application administering account data as shown in fig. 1. It is shown as a class diagram in a notation similar to UML. Directed lines denote the possibility of invocations between the classes. Folder symbols denote subsystems. The example system contains three subsystems: I/O (graphical input and output), Control and Database. For brevity, we only show the internal classes of the latter.

![Diagram of the example system](image)

**Figure 1:** Attaching subsystems to components

But how should we distribute the system in question? A common choice is to separate the database in a server process. Another possible configuration would leave only I/O on the client, Control might be executed in an own process (yielding a three tier system).

If we divide a larger application into different subsystems, it its sensible not to decide for each class to which server process (or component) it should belong. We therefore take subsystems as candidates for components. This is done by *attaching* subsystems to a component. In the figure this is denoted by dotted lines between these two. The computation of the set of classes participating in a component can be derived from the attachment relations.

Please note that this computation is not as trivial as it might first seem. Some classes must be co-located to their callers (e.g. classes providing access to sensoric data or stubs generated by an IDL compiler). They have to be included in each component they are called from. The developer should be able to provide the “co-location” information in form of an annotation.

**Distribution Structure Changes** Consider that the system at hand was previously designed with a centralized module Control located on the server component. The developer used a global variable to access an instance of the singleton object Admin. He/she also employed the new operation to create instances of Account.

Both methods are no longer valid, if Control lives not in the same address space as the database subsystem. Thus, we have to *restructure* the system in a way that it becomes immune against changes of the distribution structure.

To sum up, a developer wants to be supported by a tool in the following ways:

- The tool should allow to specify the distribution structure on a high level.
- It should capture situations that conflict with the current distribution structure.
- It should help the developer to carry out structural changes (i.e. architectural transformations).

The next section sketches the methodology and the inner workings of such a tool, which we implemented based on graph transformation technology.

### 3 Methodology and Tool Support

Up to now, we have not mentioned how the tasks outlined in the preceding section relate to graph gram-
The implementation of our tool employs a graph, representing the architecture of the subject system. A transformation of the program is thus identical to a rewriting of the architecture graph and additional source code alterations. It is also possible to gain information about the system (and potential pitfalls upon distribution) by analyzing the graph.

This tool is implemented using the high level specification language PROGRES [Sch96, SWZ95]. PROGRES features a language to specify a schema and transformations that operate on graphs conforming to the schema.

It is not only a specification language for graph transformations, but it also contains an environment which allows to edit and interpret the specification. Besides the interpretation in the PROGRES environment, a prototype can be generated (C code, user interface based on Tcl/Tk).

The PROGRES schema (which can be roughly compared to the notion of a schema in database modeling) specifies the structure of graphs. The nodes in a graph are instances of types that are specified in the schema. PROGRES uses a two tiered type system: Nodes are instances of node types which inherit their properties from one node class. Multiple inheritance is possible between node classes. A node class specifies the attributes of nodes and the (typed) edges between them.

The schema serves as a meta model for our architecture description language. It specifies the attributes and connections a node may have. It is not surprising that the graph schema “looks” similar to parts of the UML meta model [Rat97]. Figure 3 shows the schema of our architecture language. It stores the annotations in string sets (attribute of ADLObject). Inheritance relationships are denoted by dashed arrows, solid arrows represent labeled edges.

The tool consists of four coarsely integrated parts: a parser, an architecture and source transformation tool and a generator. The structure is shown in fig. 2, let us summarize the steps:

1. We gain an architectural view of a software system by analyzing the source code and building up a graph structure (reverse engineering). The analyzers we have built either use the transformation language TXL [CCH95] or the utility JavaCC (which is similar to lex/yacc). They are able to parse Java, C++ and Modula-3 programs (with some restrictions) and build up a graph representing the system’s structure.

2. In this step, we attach types and subsystems to components (“planning of distribution”). We want to increase flexibility and reusability by changing properties of types and invocations through annotations. The following list contains the annotations that are currently supported by our system.

   - affiliation to (one or more) components: This information is derived from the attachment relation of the subsystem it is embedded in.
   - local: Instances of the class should reside in the address space of their callers (true for stubs).
   - call-by-value: Instances of this class should always be passed by value (CORBA will support call-by-value in an upcoming release, see for example [Gra96]).
   - singleton: Only one instance of this class may exist. This annotation is important...
for distribution because a generated static method named e.g. `instance`, can perform nameserver lookups to bind the instance automatically.

- stateless: Instances of this class can be replicated without worrying about consistency.

We then apply graph queries to check whether (distribution) preconditions are satisfied. Take for example the requirement that caller and callee must be colocated either because of a creation (see factory example below) or a user annotation to eliminate efficiency bottlenecks. For lack of space, we omit a description of this query here.

If we want to perform a persistent transformation of the system, we have to do it on two levels: source code and architecture. The latter is described by a graph rewriting rule, the former by a sequence of basic source code transformations, broken down to basic transformations like adding, renaming, deleting classes or methods. The source code transformations are specified on a high level using the rewriting language TXL.

Let us now look at a graph rewriting rule. Fig. 4 shows a PROGRES rule that causes callers of a class to submit invocations via an interface rather than addressing a class directly. This and the following transformation aim to decouple the dependency of callers to a specific implementation.

The left hand side of the rule matches the subsystem in which the class is contained. It also checks whether the class is not already implementing an interface that follows a simple postfix naming convention (the outlined arrow denotes this restriction of an attribute value).

The right hand side introduces a new interface and an implements relation between original class and the interface. The newly created interface belongs to the same subsystem and should have identical methods to the original class. The copy clause replicates all `-has_method->` edges from the original class. All invocations of the class have to be redirected to the interface. This is handled by the embedding clause that redirects the targets of all `-calls->` edges from node `1` to the new node `4` (the newly created interface). Finally, the transfer clause assigns a new name to the interface.
Fig. 4: Introduce explicit interface declarations

Fig. 5 shows a further transformation that can be prepared by the preceeding rule: the introduction of an abstract factory. Factories are necessary in distributed systems because they allow to create objects in foreign address spaces. A factory is a “normal” object that provides a creation service through one of its methods. Instead of creating an object directly via a language primitive (denoted by a creates edge in the architecture graph) a class implementation calls the factory’s method.

The preconditions of this rule (left hand side) states that there are no direct invocations of the class for which we want to provide a factory. The factory –inserted by the right hand side of the rule– has a method create, whose implementation instantiates the desired class and returns a reference to the caller. The embedding clause replaces create-> by calls-> edges and redirects the targets to the factory (node 3’). The transfer clause initializes the attributes of the factory with proper values.

Prior to compilation, the source code is analyzed and annotations are resolved. For each component, the necessary parts of the overall application code code (computed by a graph

query) together with generated proxys and a generated makefile is written to a separate directory of the file system. We call this step a transient transformation, because the generated code is not the basis of further deployment. We do not want to keep this code, as it is middleware dependent.

For each class that is remotely accessible, the proxy stores a reference to the stub generated by the IDL compiler. Instead of calling CORBA’s stubs directly we indirect invocations via our proxy to the stub. The generator can thus insert annotation related code in the proxy before and after the delegation to CORBA’s stub. On the server side we use another proxy to decouple the original implementation from CORBA specific code. The code in the proxy can be customized by template files.

The proxy generator has been written completely in Java (about 10.000 lines of code) as part of a diploma thesis. Currently, we support the generation of Java proxies.
shown in figure 6. The graph represents the architecture of our running example.

4 Related Work

Pattern Rewriting  ZÜNDORF and JAHNKE [JZ97] apply graph rewriting rules (incorporating a pattern) to transform architectures. They use a fuzzy reasoning. They also incorporate other technologies into their tool: they use a fuzzy reasoning net to detect “bad code” in existing programs. Another pattern oriented transformation tool has been developed by FLORIJN et al. [FMW97]. Their system allows to bind existing classes to a role in a pattern or to create new classes by instantiating a pattern. The tool is not based on graph technology.

Our tool differs from these two because of (1) the focus on distribution aspects, and (2) the distinction between persistent and transient transformations.

Architectural Tools  UML also uses a metamodel to specify terms of the modeling language, e.g. classifier as a base class for class and interface. Most existing UML tools feature roundtrip engineering, but have no support for the creation of distributed systems besides the generation of CORBA IDL from class definitions.

Annotations  One idea of our approach is the annotation of the architecture. This could be interpreted as a specific aspect language [KIL+96]: The annotations provide information about certain aspects of the invocation (e.g. synchronization issues) or about a type, i.e. a class. Some annotations are simply boolean properties, others —like synchronization— require a slightly more complex definition. As in LOPES’ framework “D” [LK97], we must be able to describe locks on the level of methods of a type.

Implementation of Annotations  There are different ways of implementing the annotations. There has been work on the use of reflection [BMR+96] and meta object protocols [LL96]. A meta object protocol allows us to inspect and adapt the way objects are handled in an object oriented programming language. Here, we are mostly interested in an interception of a method call. MOPs are not supported.
in most programming languages. If this concept is not inherent to the programming language, extended variants of typical programming languages must be used. The lack of availability of such a language extension for many platforms and languages is a major disadvantage of this approach.

As the interception of a method call is probably most important, filters that do specifically this job can be used. This idea has been investigated from the TRESE group, using the specific programming language SINA and the composition filters [AWB+93] approach. Interception has recently become a standardized feature or CORBA. In order not to be restricted to CORBA, we have chosen to generate proxies.

## 5 Summary

We have shown that the task of distributing an existing application via CORBA can be tackled using a suitable tool and flexible techniques (connector properties) that avoid an inflation of the application code with distribution details.

The novelty of our approach is the use of graph techniques for analysis and transformation of architectures. The graphical specifications found in step 1 have the advantage that they provide an intuitive, yet well defined means to document and execute architectural transformations. Preconditions of distribution (e.g. a missing factory, see above) are explicitly modeled and can be checked by analyzing the graph structure, i.e. by specifying a graph query. Queries enable us to examine more complex properties of classes and their relationships, i.e. object-oriented design patterns. This information might put us in the position to employ efficient strategies e.g. for migration and replication.

The developed tool combines graph rewriting with compiler technology: it also requires parsing, source code transformation, and generator techniques to analyze source code, to keep it consistent with the transformations at the architecture level and to generate code that implements connector properties, respectively. Currently, parser and generator are loosely coupled with the prototype generated by the PROGRES environment (via scripts and a file describing the architecture). In the future, we plan a more tight integration of the prototype with other tools, probably via a CORBA interface.

## References


[KIL+96] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Christina Videira Lopes, Chris Maeda, and Anurag Mendhekar.


